Interim - Zwischen Plan 9 und Lisp Machine

Vasilij Schneidermann

Juni 2017

Outline

Abschnitt 1

Einführung

Sprecher

- Vasilij Schneidermann, 24
- Wirtschaftsinformatikstudent
- Software-Entwickler bei bevuta IT GmbH
- mail@vasilij.de
- https://github.com/wasamasa
- http://emacshorrors.com/
- http://emacsninja.com/

Grundlegendes Problem

- Moderne Computer sind komplex
- Unmöglich den gesamten Stack zu verstehen
- Walled Gardens breit akzeptiert
- Kontrollverlust über das Betriebssystem
- Wird es jemals besser?

Damals war alles besser™

Abbildung: Hackerman (Kung Fury)

Damals war alles ~~besser™~~ einfacher

- Bestseller: Commodore 64 (1982)
- Boot in einen BASIC-Prompt
- Umfangreiches Handbuch für Reparaturen
- Populär in der Demoszene
- Verwendet für BBS
- Später durch den Amiga verdrängt

Wie würde ein moderner C64 aussehen?

- RISC-Prozessor
- High-Level Programmiersprache
- Benutzeranpassbares Betriebssystem
- 1920x1080 ansteuerbare Pixel in 16-/24-Bit Farbe
- Keyboard und mausgesteuerte Benutzereingabe
- Audio-Output in CD-Qualität
- Netzwerkanbindung über Ethernet

Wie würde ein einfacheres Betriebssystem aussehen?

- Plan 9
- Lisp Machine
- Project Oberon

Motivation für diesen Vortrag

- The Future of the LispM
- Beschreibung einer Alternative zur Lisp Machine:
  - Betriebssystem mit JIT-Compiler
  - Moderne Lisp-Implementierung
  - Plan 9 statt Unix
- Interim erfüllt diese Kriterien und mehr

Abschnitt 2

Interim - Aufbau

Vorab

- Sämtliche Aussagen über Interim beziehen sich auf meinen Fork auf `https://github.com/wasamasa/interim/tree/next`
- Enthält zusätzliche Dokumentation, Bugfixes und Features

Betriebsmodi

- Typischerweise OS für eine Zielplattform entwickelt
- Testen im Emulator oder auf echter Hardware
- Interim unterstützt *Hosted Mode* und *Bare Metal*

Hosted Mode

- Lisp-Interpreter als Kern
- Kann auf POSIX-kompatiblem System mit `libc` ausgeführt werden
- Nutzt SDL2 für Maus, Tastatur und Framebuffer
- Läuft auf Linux, Windows, OS X
- Experimenteller Support für AmigaOS

Bare Metal

- Portierung des Interpreters auf System ohne `libc`
- Verwendung von newlib als `libc`
- Assembler für Setup des *Stacks* und Boot ins Programm nötig
- Systemspezifische Geräteabstraktionen
- Läuft auf Raspberry Pi 2
- Experimenteller Support für regulären x86-PC

Boot (Hosted Mode)

- Init des JIT-Compilers
- Init von Dateisystemen
- Einlesen einer Datei
- Alternativ: Start der REPL

Boot (Bare Metal)

- BSS-Sektion initialisieren
- Stack-Setup
- Start des Kernels
- Hardware-Setup (MMU, UART, Framebuffer, USB, . . .)
- Init des JIT-Compilers
- Init von Dateisystemen
- Start der grafischen REPL

Device-Abstraktion

- Idee: Gleiche API für ähnliche Hardware
- Implementierungen können grundverschieden sein
- Erlaubt Entwicklung in Hosted Mode und Testen auf Bare Metal
- Beispiel: Keyboard in `/devices/sdl2.c`, `/devices/rpi2/uartkeys.c` und `/devices/rpi2/usbkeys.c` implementiert

Compiler

- Problem: Naiver Interpreter zu langsam, AOT-Compiler nicht anwendbar
- Lösung: Inkrementeller JIT-Compiler (vgl. Tracing JIT-Compiler)
- Abstraktion von ISA-spezifischen Instruktionen (x86, amd64, m68k, arm64)
- Kompilieren von Lisp zu diesem Instruktions-Set
- Schreiben des Codes in ausführbaren Speicher
- Cast zu Funktionspointer und Aufruf
- Probleme: Calling Convention, Memory Barriers, Debugging schwierig

"Multitasking"

- Single-tasked
- Emulation von kooperativem Multi-Tasking
- Liste von Tasks (Funktionen)
- Wiederholte Iteration über Taskliste

Abschnitt 3

Interim - Sprache

Grundeigenschaften

- Es ist ein Lisp!
- Homoikonisch, minimalistisch, high-level
- Features: Globale/lokale Variablen, Integer-Arithmetik, einfache Kontrollstrukturen, Listen/Array-Manipulation, Introspektion, Dateisystemzugriff
- Typen: Integers, Strings, Byte-Arrays, Funktionen, Listen, Structs
- Manuelle Garbage Collection

Unterschiede zu anderen Lisp-Dialekten

- `let` ohne Body, nur in Funktionen zulässig
- `do` ist nie implizit
- Keine Makros oder Fexprs
- Kein syntaktischer Zucker (Readermakros)
- Keine Booleans (siehe C!)
- Serialisierung nur mit fixen Buffern möglich (kein `str`)
- Minimale Standardbibliothek (beinhaltet `or`, `strlen`, `sin`, ...)

Beispiele

```
(def greeting "Hello World!")
(print greeting) ;=> "Hello World!"

(put8 greeting 11 (get8 "?" 0))
(print greeting) ;=> "Hello World?"
```

Beispiele

```
(+ 1 1) ;=> 2
(cons 1 2) ;=> (1 . 2)
(list 1 2) ;=> (1 2)
(cons 1 (cons 2 nil)) ;=> (1 2)

(def bytes [1234])
(put8 bytes 0 0x34)
(put8 bytes 1 0x12)
bytes ;=> [3412]
```

Beispiele

```
(def factorial
 (fn n
  (do
   (let i 1)
   (let result 1)
   (while (lt i n)
    (do
     (let i (+ i 1))
     (let result (* result i))))
   result)))

(factorial 4) ;=> 24
```

Abschnitt 4

Interim - Dateisysteme

Plan 9-Dateisysteme

- Jedes Gerät wird unter einem Pfad gemountet
- Mounten erforder folgende Handler:
  - `open`: Öffnen eines Stream-Objekts für den gegebenen Pfad
  - `mmap`: Anfordern einer alternativen Repräsentation des Pfads
  - `recv`: Auslesen eines Objekts aus dem gegebenen Stream
  - `send`: Schreiben eines Objekts in den gegebenen Stream

Beispiel: `/framebuffer`

- Implementierung: `/devices/sdl2.c`, `/devices/fbfs.c`, `/devices/dev_linuxfb.c`
- `open`: Öffnen eines Kontrollkanals für den *Framebuffer*
- `mmap`: Anfordern des *Framebuffers* in Form eines Byte-Arrays
- `recv`: Gibt Liste von Attributen oder Attribute selbst aus
- `send`: Löst eine *Blit*-Operation aus

Beispiel: Framebuffer-Parameter

```
(def fb (open "/framebuffer"))
(def refresh (fn (send fb 0)))
(def load (fn path (recv (open path))))

(def width (load "/framebuffer/width"))
(def height (load "/framebuffer/height"))
(def depth (load "/framebuffer/depth"))

(def pitch (* width depth))
(print (* height pitch)) ;=> 960000
```

Beispiel: Framebuffer-Manipulation

```
(def pixels (mmap "/framebuffer"))
(def black 0x0000)

(def paint-pixel
 (fn x y color
  (do
   (let offset (+ (* y pitch) (* x depth)))
   (put16 pixels offset color))))

(paint-pixel 0 0 black)
```

Beispiel: `sledge/demos/palette.l`

- Idee: Framebuffer erlaubt 2^{8*bpp} Farben
- Bei 2bpp: Farbwerte zwischen 0 und 65535
- Quadrat mit jedem Farbwert: Palette

Beispiel: `/sd`

- Implementierung: `/devices/posixfs.c`, `/devices/fatfs.c`
- `open`: Nicht implementiert
- `mmap`: Nicht implementiert
- `recv`: Gibt Liste von Verzeichniseinträgen oder Dateinhalt zurück
- `send`: Schreibt in eine Datei

Beispiel: Laden eines Bilds

Vorbereitung:

```
ffmpeg -i image.jpg -vcodec rawvideo -f rawvideo\
  -pix_fmt rgb565 image.565
```

Laden:

```
(def load (fn path (recv (open path))))
(def image (load "/sd/image.565"))
```

Beispiel: `sledge/demos/grumpycat.l`

- Kenntnis von Breite und Höhe nötig
- *Image Loader* ist nicht nötig
- Jeder Bildpixel wird an die richtige Stelle positioniert

Beispiel: `sledge/demos/helloworld.l`

- Fontformate sind sehr komplex
- Alternative: Speichern von GNU Unifont als Bitmap
- Position jedes Zeichens ist errechenbar
- Kopieren von Zeichen auf richtige Position am Framebuffer
- Sogar Cursor so implementierbar!

Beispiel: `sledge/demos/screenshot.l`

- Roher Screenshot ist trivial
- Screenshot zu Format tricky wegen Kompression, Pixelformaten
- BMP und TGA sind die einfachsten Formate, aber:
  - TGA unterstützt kein kompatibles Pixelformat
  - BMP ist unterspezifiziert, wird je nach Viewer verschieden angezeigt

Beispiel: `sledge/demos/munchingsquares.l`

- Klassische Demo von 1962
- Animation besteht aus Zeichnen aller Pixel und Warten
- Warten durch Ausführen von `(gc)` implementiert
- Algorithmus: Für Frame n zwischen 1 und 16:
  - x xor $y < n$
  - Wenn wahr, ist der Pixel schwarz, sonst weiß

Beispiel: `/keyboard`

- Implementierung: `/devices/sdl2.c`, `/devices/rpi2/uartkeys.c`, `/devices/rpi2/usbkeys.c`
- `open`: Nicht implementiert
- `mmap`: Nicht implementiert
- `recv`: Gibt aktuell gedrückte Taste oder `nil` zurück
- `send`: Nicht implementiert

Beispiel: `sledge/demos/bounce.l`

- Zeichnen eines Quadrats an einer Position
- Übermalen des Quadrats an alter Position in weiß
- Errechnen einer neuen Position
- Ändern der Richtung bei Erreichen der Kante
- Bonus: Drücken von Tasten ändert die Farbe

Weitere nicht abgedeckten APIs

- Maus (`/mouse`)
- Netzwerk (`/net`)
- Implementierung eigener Dateisysteme aus Lisp heraus

Abschnitt 5

Weitere Schritte

Verbessern der Sprache

- First-class functions
- Makros und Readermakros
- Automatische *Garbage Collection*
- Mehr Datentypen (Vector, Hash Map)
- Prädikate, mehr Introspektion
- Exceptions

Komfortfeatures

- Mehr Fehler signalisieren
- Optionale Argumente / Varargs / `apply`
- Escapes in Strings
- Besserer Printer und Print-Funktionen
- Debugging-Funktionalität
- Ersetzen von `fn` / `while` durch Makro mit `do`

Testsuite für Sprachfeatures

- Die aktuelle Dokumentation ist nicht auf dem neuesten Stand
- Viele Demonstrationsprogramme welche nicht mehr funktionieren
- Einige Features sind kaputt (teilweise nur auf einer Plattform)
- Integration von CI

APIs

- Verhalten in *Hosted Mode* an *Bare Metal* anpassen
- Implementierung der `delete`-Operation
- Implementierung weiterer APIs:
  - `/arch`, `/sys` (Systeminformationen)
  - `/time` (kein `sleep` bisher möglich)

Verbessern der Demos

- Spiele (`mario.l`, `gtn.l`)
- Grafische Shell und Editor (`shell.l`, `editor.l`)
- Netzwerk (HTTP, IRC)

Bugs beheben

Kleine Auswahl:

- GC rührt lokale Variablen an
- *Undefined Behavior*
- Segfault bei Verwendung von `put32`
- Off-by-one in String-Funktionen
- Minimum an Fehlerbehandlung

Bauen eines besseren Interim

- Vielleicht ist die einfachste Lösung von neuem anzufangen...
- Ermöglicht neue Design-Entscheidungen:
  - Byte-Code Interpreter statt JIT-Compiler
  - Wiederverwendung von Libraries (Ragel, QBE, ...)
  - Bessere Lisp-Implementierung
  - Alternative zu C

Fragen?